L2 biologie
Gilles Bernot

Cours d'Informatique
Notes de cours

année 2005-2006
Université d'Évry

Avertissement au lecteur :

Ce polycopié n'est pas un document scolaire de référence sur le cours d'informatique, c'est seulement l'ensemble de mes propres notes de cours mises en forme. Il ne contient donc pas les explications détaillées qui ont été données en cours. En particulier tous les développements systématiques des exemples, expliquant comment le langage ML effectuerait les traitements, sont absents de ces notes. On trouvera également parfois quelques simples allusions à des concepts élémentaires, largement développés en cours mais qui sont routiniers pour tout informaticien.

G. Bernot

COURS 1

1. Rappels sur le langage ML
2. Rappels sur les types de base
3. Rappels sur le contrôle

COURS 2

1. Rappels sur les types de données colposés
2. Rappels complémentaires sur le contrôle

COURS 3

1. Les constructeurs d'un type de données
2. Les types « somme »

COURS 4

1. Usage de `match` sur les types somme

COURS 5

1. Les structures d'arbres binaires
2. Fonctions classiques sur les arbres binaires

COURS 6

1. Autres fonctions sur les arbres binaires
2. Les stratégies de parcours d'un arbre binaire

COURS 7

1. Les « mots » en informatique
2. La recherche de motif (exact) dans une séquence

COURS 8

1. Recherche de toutes les occurences d'un motif
2. Recherche de motif avec un seul parcours
3. Les automates simples
4. Utiliser un automate
5. Construire un automate

COURS 9

– Recherche de motif exact en général

COURS 10

– Alignement de séquences, motifs approchés

COURS 11

– Les fonctions d'ordre supérieur
– Exemples de fonctions d'ordre supérieur
– Fonctions d'ordre supérieur sur les listes

COURS 12

– La curryfication

Cours No. 1

Rappels sur le langage ML

Un langage est défini par les structures de données qu'il offre et par son contrôle. Un langage fonctionnel comme ML possède peu de dispositifs de contrôle car la notion même de fonction qui est à la base du langage offre gratuitement presque toutes les structures de contrôle utiles. On fonde donc cette révision sur les structures de données et le contrôle sera vu au passage en fonction des besoins.

Les types de données permettent de classer et caractériser proprement les structures de données en ML. On passe donc en revue les différents types de données vus l'an dernier. On commence par les types de base, c'est-à-dire ceux qui ne servent pas à construire de nouveaux types de données à partir d'autres types de données. On continuera par les types de données composés, c'est-à-dire ceux qui se construisent de manière générique à partir d'un ou plusieurs autres types de données quelconques.

Rappels sur les types de base

On révise rapidement les types suivants
- Les booléens (`bool`).
- Les nombres entiers relatifs (`int`).
- Les nombres réels (`float`)
- Les chaînes de caractères (`string`)

Rappels sur le contrôle

On profite de ces rappels pour donner des exemples et à cette occasion on revoit les éléments de contrôle suivants :
- Les déclarations avec `let`.
- Les fonctions récursives avec `let rec`.
- Les conditionnelles `if..then..else..` qui peuvent en fait être vues comme une fonction générique de type $\texttt{bool} \times \alpha \times \alpha \rightarrow \alpha$.

L2 biologie
Gilles Bernot

Cours No. 2

cours d'informatique

1 Rappels sur les types de données composés

Une structure de données est définie par 2 choses :

1. un ensemble, dont les éléments sont les *données* de la structure de données ; et le nom de cet ensemble est appelé le *type* de ces données
2. des *fonctions* (au sens mathématique du terme), que l'on appelle aussi les *opérations* de la structure de données car elles sont utilisées par l'ordinateur pour effectuer des « calculs » sur les données précédentes.

On revoit avec plusieurs exemples les deux types de données suivants :

- Les listes dont les éléments sont d'un type quelconque α, c'est-à-dire les α `list`.
- Les types enregistrement.

On donne en exemple la gestion d'une liste d'étudiants, chaque étudiant ayant un `nom`, un `numero` de carte et une `moyenne`.

2 Rappels complémentaires sur le contrôle

À cette occasion on révise :

- L'usage de `match` sur les listes
- Le traitement d'exceptions avec `failwith`

Ceci termine les révisions.

Le TD correspondant sera un examen de L1, ce qui permet de tester les connaissances en général.

Cours No. 3

On aborde maintenant une nouvelle manière de définir de nouveaux types de données en ML : les *types « somme »*. Tout d'abord, il nous faut bien comprendre la notion de constructeurs d'un type de données.

1 Les constructeurs d'un type de données

Certaines constantes et fonctions jouent un rôle particulier au sein d'un type de données car elles permettent de construire inductivement toutes les données de ce type.

Dans le cas des listes par exemple, la constante `[]` et l'opération `_::_` permettent de construire toutes les listes. En effet, étant donné un type de données α quelconque, supposé déjà fourni, on peut construire toutes les listes avec ces deux fonctions :

- la constante `[]` fournit la seule liste de longueur 0 qui existe ;
- on peut obtenir toutes les listes de longueur 1 en utilisant l'opération `_::_` comme suit : `a::[]` et en considérant toutes les valeurs possibles de `a` au sein du type α ;
- plus généralement, on peut obtenir toutes les longueurs de liste possibles en utilisant `_::_` autant de fois que necessaire (longueur de la liste) : `a_1::a_2::...a_n::[]`

On fait le lien entre cette remarque et l'usage de l'opérateur `match` qui décompose les listes selon ce principe exactement. C'est le fait que ces deux fonctions sont les constructeurs choisis pour les listes qui autorise à faire `match` de cette façon dans les programmes.

On pourrait tout à fait définir le type des entiers naturels, de manière encore plus simple, avec les constructeurs 0 et `succ_`, où l'opération successeur est intuitivement celle qui ajoute 1 à son argument. Tout entier naturel est obtenu en appliquant `succ` autant de fois que nécessaire à 0. En fait, par définition, l'entier n est égal à n applications successives de `succ` au-dessus de 0.
D'un point de vue purement mathématique, l'ensemble des entiers naturels, $\mathbb{N}$, est défini par les axiomes de Peano :

1. 0 existe
2. si n existe alors `succ`(n) existe
3. `succ` est injective
4. 0 n'est le successeur d'aucun entier naturel
5. quelle que soit la propriété P, pour prouver $\forall n \in \mathbb{N},\ P(n)$ il suffit de prouver $p(0)$ et $\forall i \in \mathbb{N},\ P(i) \Rightarrow P(\texttt{succ}(i))$

Les deux premiers axiomes disent exactement que 0 et `succ` sont constructeurs de $\mathbb{N}$. Le cinquième n'est autre que la récurence et dit en réalité qu'il n'y a pas d'autre entier naturel que ceux *engendrés* par 0 et `succ`. Enfin le troisième et le cinquième évitent des structures de données circulaires. Ces axiomes fondamentaux de l'arithmétique traduisent en fait les fondements mathématiques des fonctions récursives.

Les entiers relatifs sont un exemple intéressant. On a en tête facilement deux ensembles de constructeurs « raisonnables » : $\{0, succ, pred\}$ et $\{0, succ, opp\}$. Cet exemple nous permet d'insister sur le fait que plusieurs choix de constructeurs peuvent être faits pour une même structure de données. Ici on constate qu'il y a des « équations entre constructeurs », à savoir, d'une part, $succ(pred(n)) = n = pred(succ(n))$ et, d'autre part, $opp(0) = 0$ et $opp(opp(n)) = n$.

On remarque donc que le choix des constructeurs d'un type de données n'est pas unique : cas d'école liste vide `[]`, liste de taille 1 `[_]` et concaténation `@`. Certains choix de constructeurs entrainent des équations entre constructeurs parfois lourdes et c'est souvent cela qui donne une préférence pour un choix de constructeurs plutôt qu'un autre.

Enfin il existe un cas particulier de types de données (fini) où les constructeurs ne sont que des constantes. C'est le cas des types `bool` et `char`. On aura envie bien sur de définir d'autres types de ce genre comme les nucléotides `{A,T,G,C}`, les couleurs `{R,V,B}`, etc.

2 Les types « somme »

Comment on les déclare :

```
type α_1 ··· α_n truc =
      cst_0 | ... | cst_k
     | c_1 of τ_1 | ... | c_p of τ_p ; ;
```

où les τ_j peuvent non seulement contenir les α_i mais aussi de manière récursive le type (α_1 $\cdots$ α_n `truc`) lui-même.

Exemples :
- nucléotides
- couleurs
- une redéfinition des listes
- arbres binaires : avec deux possibilités
  - admettre qu'un arbre *vide* est une donnée acceptable
    ```
    type 'a Abin = vide
                 | node of 'a * ('a ABin) * ('a ABin) ;;
    ```
  - ne pas accepter d'arbres vides, auquel cas il faut distinguer les noeuds n'ayant qu'un seul fils
    ```
    type 'a Abin = feuille of 'a
                 | birac of 'a * ('a ABin) * ('a ABin)
                 | monorac of 'a * ('a ABin) ;;
    ```

Cours No. 4

1 Usage de `match` sur les types somme

Forme générale, utilisée dans une expression :

... (`match` *expr* with
$motif_1 \rightarrow result_1$
$\mid motif_2 \rightarrow result_2$
$\mid \ldots$
$\mid motif_n \rightarrow result_n$)
...

où les motifs sont des termes qui ne peuvent faire intervenir que des variables ou des constructeurs de types. On comprend mieux sur des exemples...

Les nocléotides :

```
type base = A | T | G | C | N ;;

let complementaire b = match b with
  A -> T | T -> A | G -> C | C -> G | N -> N ;;
```

Les listes :

```
let rec est_compl (l1,l2) = match (l1,l2) with
      ([],[])      -> true
 |     ([],_)      -> false
 |     (_,[])      -> false
 | (b1::r1,b2::r2) -> (b2 = (complementaire b1))
                       && est_compl(r1,r2) ;;
```

On remarque l'usage de « _ » qui remplace une variable inutilisée. La faiblesse de cette solution est que `N` devrait être considéré comme complémentaire de n'importe quelle base dans le cas de la comparaison de brins :

```
let rec est_compl (l1,l2) = match (l1,l2) with
      ([],[])      -> true
 | (N::r1,b2::r2)  -> est_compl(r1,r2)
 | (b1::r1,N::r2)  -> est_compl(r1,r2)
 | (b1::r1,b2::r2) -> (b2 = (complementaire b1))
                       && est_compl(r1,r2)
 |     _      -> false
```

On rappelle que le premier motif qui généralise la valeur traitée est celui qui sera choisi losque la fonction est exécutée. Ainsi l'avant dernière ligne n'a lieu que si le premier nucléotide des deux brins est connu (différent de `N`). De même la dernière ligne généralise les deuxième et troisième motifs de la version précédente.

Les piles (`stack`) : constructeurs `emptystack` et `push`. fonctions `isEmpty`, `top`, `pop`, `height`. On remarque que c'est un sous-ensemble des choses que l'on peut faire avec les listes mais on mentionne rapidement les questions d'encapsulation qui justifient l'existence de types plus pauvres que les autres.

Les files FIFO : constructeurs `vide` et `ajout`. Fonctions `longueur`, `premier`, `supprime`. En insistant sur la technique des appels récursifs, et ne pas oublier de remettre les éléments non traités pour `supprime`.

Cours No. 5

1 Les structures d'arbres binaires

Les arbres binaires standard :

```
type 'a ABin = empty
 | node of ('a ABin) * 'a * ('a ABin) ;;
```

Les arbres binaires complets :

```
type 'a ABComp = feuille of 'a
 | noeud of ('a ABComp) * 'a * ('a ABComp) ;;
```

et on développe des exemples d'arbres (dessin abstrait et expression ML correspondante) pour chacun des types présentés.

2 Fonctions classsiques sur les arbres binaires

On se restreint ici au premier type (qui est en fait plus général). On développe les fonctions classsiques : `taille`, `hauteur` (2 versions selon qu'une feuille est de hauteur 1 ou 0), `nbFeuilles`.

Cours No. 6

1 Arbres binaires : exemples d'autres fonctions

`sup` et `inf`, `somme`, etc.

```
let rec sup a = match a with
    empty -> failwith "arbre vide!"
  | node(r,empty,empty) -> r
  | node(r,empty,d) -> sup d
  | node(r,g,empty) -> sup g
  | node(r,g,d) -> max(r, max( (sup g),(sup d) ) ) ;;
```

et toutes les remarques sur le message d'erreur que fournirait la fonction si l'on oubliait l'un des cas avec fils vide. + développement d'un exemple.

```
idem pour inf (avec min)
```

Remarquer qu'il n'est pas nécessaire d'aller aussi loin dans la décomposition du `match` dans tous les cas. Il faut bien comprendre pourquoi l'on décompose d'une manière ou d'une autre un type de données :

```
let rec somme a = match a with
    empty -> 0
 | node(r,g,d) -> r + (somme g) + (somme d) ;;
```

En général, on peut effectuer ce genre d'*itération* pour n'importe quelle opération assiciative possédant un élément neutre.

```
let rec produit a = match a with
    empty -> 1
 | node(r,g,d) -> r * (somme g) * (somme d) ;;
```

... ou bien « `0.0` » et « `*.` » s'il s'agit de réels, etc.

2 Les stratégies de parcours d'un arbre binaire

Lorsque l'on applique le schéma de programmation précédent à des opérations non commutatives (contrairement à l'addition, la multiplication mais aussi le sup et l'inf), l'*ordre d'énumération* de l'arbre compte. On va donc étudier divers parcours d'arbres binaires. Pour cela on supposera vouloir faire une enumeration des éléments de l'arbre dans une liste.

En profondeur d'abord, à gauche d'abord :

```
let rec enum1 a = match a with
    empty -> []
  | node(r,g,d) -> (enum1 g) @ (enum1 d) @ [r] ;;
```

Attention au typage correct avec les `@` et les `: :` !

La racine d'abord, puis à gauche et en profondeur (i.e. racine d'abord) :

```
let rec enum2 a = match a with
    empty -> []
  | node(r,g,d) -> r::(enum2 g) @ (enum2 d) ;;
```

Idem avec « à droite » au lieu de « à gauche ».

En largeur d'abord... Si l'on tente le `match` direct, on ne réussit pas la programmation :

```
let rec enum3 a = match a with
    empty -> []
  | node(r,empty,d) -> r::(enum3 d)
  | node(r,g,empty) -> r::(enum3 g)
  | node(r,node(rg,gg,dg),node(rd,gd,dd)
                           -> r::rg::(...???...
```

On ne réussit pas car il faudrait « matcher » sur `gg` de la même façon qu'on l'a fait sur `a` puis `g` et `d` et le nombre de cas à écrire deviendrait infini car il faudrait recommencer avec tous les fils des fils des fils...

Cela signifie que l'on n'adopte pas le bon principe de récurence pour énumérer en largeur. On constate que pour énumérer correctement, il faut écrire la racine du sous-arbre en cours de traitement et mémoriser qu'il faudra traiter chacun de ses fils lorsqu'on aura fini de traiter ce qui a déjà été mémorisé. Bref, il faut traiter à chaque ionstant le sous-arbre le plus vieux parmi ceux qui ont été mémorisés. Il faut donc gérer une file FIFO d'arbres à énumérer. Dès lors, on réutilise le type fifo

```
type 'a fifo = vide
  | ajout of 'a * ('a fifo) ;;
```

pour lequel on avait programmé les fonctions `longueur`, `premier`, `supprime`.

```
let rec fifoEnum f = match f with
   vide -> []
 | _ -> ( match (premier f) with
            empty -> fifoEnum (supprime f)
          | node(r,g,d)
             -> r::(fifoEnum ajout(d,
                                   ajout(g,
                                         (supprime f)
                                        )
                                  )
                    )
        );;
```

Le fait de changer de structure de données (des arbres aux files FIFO) nous a offert le bon principe de récurence. Il ne reste qu'à écrire la fonction demandée :

```
let enum4 a = fifoEnum (ajout(a,vide)) ;;
```

Cours No. 7

1 Les « mots » en informatique

L'objectif pragmatique en post-génomique est de trouver divers « signaux »permettant l'annotation des génomes, ou de prédire la structure d'une protéine ou d'un ARN. Néanmoins, en informatique cela relève d'un domaine plus ancien et plus général : la « théorie des mots ».

On se fondera ici sur un ensemble restreint de notations et définitions :

1. On se donne un ensemble de « symboles » S qu'on appelle l'*alphabet*. Chacun de ses élément est souvent appelé une *lettre*.
   Exemple : on peut choisir l'ensemble des nucléotides $S = \{A, T, G, C\}$, ou bien si l'on souhaite admettre une valeur indéfinie $S = \{A, T, G, C, N\}$, ou encore pour l'ARN $S = \{A, U, G, C\}$, ou bien pour les protéines $S = \{Phe, Leu, Ser, Tyr, \ldots\}$
2. L'ensemble des *mots sur l'alphabet* S est l'ensemble des séquences de lettres de S. Un mot ω est donc de la forme « $c_1 c_2 \cdots c_n$ » où n est le nombre de lettres (la longueur) du mot et chaque c_i est un élément de S. On note S^* l'ensemble des mots sur S.
   Exemple : ω=AAATGTCC est un mot de longueur 8 sur l'alphabet $S = \{A, T, G, C\}$. Le mot ω est un élément de S^*.
3. Par convention on note ε le mot vide (de longueur 0).
4. Si $\omega = c_1 c_2 \cdots c_n$ et $\omega' = d_1 d_2 \cdots d_m$ sont des mots (de longueur n et m respectivement) alors par convention $\omega.\omega'$ est la concaténation de ces deux mots : $\omega.\omega' = c_1 c_2 \cdots c_n d_1 d_2 \cdots d_m$. L'opération « . »de concaténation de mots est associative et ε est élément neutre pour cette opération : $\varepsilon.\omega = \omega.\varepsilon = \omega$
   Exemple : si ω=AAATGTCC et ω'=TCT alors $\omega.\omega'$=AAATGTCCTCT
5. Un mot τ est un *préfixe* du mot ω si et seulement si il existe un mot σ tel que $\omega = \tau.\sigma$
   Exemple...
6. Un mot τ est un *suffixe* du mot ω si et seulement si il existe un mot σ tel que $\omega = \sigma.\tau$
   Exemple...
7. Un mot τ est un *sous-mot* (ou encore un motif) du mot ω si et seulement si il existe un mot σ et un mot σ' tels que $\omega = \sigma.\tau.\sigma'$
   Exemple...
8. On remarque que τ est un motif de ω si et seulement si il est préfixe d'un suffixe de ω (ça marche aussi avec suffixe de préfixe).

2 Recherche de motif (exact) dans une séquence

Pour programmer la recherche de motif dans un mot, on commence par encoder en ML l'ensemble alphabet par un type énuméré. Par exemple :

```
type nucleotide = A | T | G | C ;;
```

On ne prend pas le `N` car nous allons faire dans un premier temps une recherche exacte de motif (selon la définition qui précède) et il est raisonnable de supposer dans ce cas que l'on connait les bases du génome étudié avec précision.

Le mot ω dans lequel on cherche un motif peut par exemple être un chromosome, ou une séquence quelconque recueillie expérimentalement. Le motif τ que l'on cherche peut être un site quelconque dont on connaît le code avec précision. Il est naturel de représenter les mots par des listes dont les éléments sont du type de l'alphabet. Ici des `nucleotide list` donc.

Dans un premier temps, on va programmer une opération simple : le préfixe.

```
let rec prefixe (tau,omega) = match (tau,omega) with
    ([],_) -> true
  | (_,[]) -> false
  | (t::tt,o::oo) -> (t=o) && (prefixe (tt,oo)) ;;
```

Ensuite, comme pour le parcours en largeur des arbres binaires, on se rend compte qu'il est difficile de programmer `motif` directement :

```
let rec motif (tau,omega) = match (tau,omega) with
    ([],_) -> true
  | (_,[]) -> false
  | (t::tt,o::oo) -> if t <> o then motif (tau,oo)
                               else ??? ;;
```

On s'en sort bien en utilisant la fonction `prefixe`, qui permet de faire `match` sur `omega` seulement :

```
let rec motif (tau,omega) = match omega with
    [] -> tau = []
  | _::oo -> (prefixe(tau,omega)) || (motif(tau,oo)) ;;
```

On peut vouloir l'indice de la première occurence :

```
let rec rangMotif (tau,omega) = match omega with
    [] -> if tau = [] then 0
          else failwith "motif absent"
  | _::oo -> if (prefixe(tau,omega)) then 1
             else 1 + (rangMotif(tau,oo)) ;;
```

Remarquer comment le message d'erreur peut être atteint à la suite d'appels récursifs même si `omega` n'est pas vide au départ.

Cours No. 8

1 Recherche de toutes les occurences d'un motif

Rappelons que pour chercher la première occurence on avait écrit :

```
let rec rangMotif (tau,omega) = match omega with
    [] -> if tau = [] then 0
          else failwith "motif absent"
  | _::oo -> if (prefixe(tau,omega)) then 1
             else 1 + (rangMotif(tau,oo)) ;;
```

On peut s'en inspirer pour `tousMotifs`

```
let rec tousMotif (tau,omega) = match omega with
    [] -> if tau = [] then [0] else []
  | _::oo -> if prefixe (tau,omega)
               then 1::(mapadd1 (tousMotif (tau,oo)))
               else mapadd1 (tousMotif (tau,oo)) ;;
```

Comme d'habitude, lorsqu'un sous-problème apparaît difficile sur une structure de données (ici ajouter 1 à tous les éléments d'une liste d'entiers), on suppose d'abord la question résolue par une fonction (ici `mapadd1`) que l'on programme à part :

```
let rec mapadd1 l = match l with
    []  -> []
  | n::r -> (n+1)::(mapadd1 r) ;;
```

Naturellement, il faut programmer `mapadd1` d'abord. On peut aussi programmer cette fonction par une déclaration locale :

```
let rec tousMotif (tau,omega) =
  let rec mapadd1 l = match l with
          []  -> []
        | n::r -> (n+1)::(mapadd1 r)
  in match omega with
      [] -> if tau = [] then [0] else []
    | _::oo -> if prefixe (tau,omega)
                 then 1::(mapadd1 (tousMotif (tau,oo)))
                 else mapadd1 (tousMotif (tau,oo)) ;;
```

On développe l'exemple `tousMotif ([A;T],[G;A;C;A;T;T;A;T;G;C])`.

Cette approche n'est pas efficace car on examine de nombreuses fois la même lettre de `omega` (jusqu'à autant de fois qu'il y a de lettres dans le motif). Pour ne parcourir qu'une fois chaque lettre, on utilise la théorie des *automates* qui permet de développer des algorithmes beaucoup plus efficaces.

2 Recherche de motif en ne lisant qu'une fois les lettres

L'idée est de ne lire qu'une fois chaque lettre de ω mais de se souvenir combien de lettres du motif τ ont été correctement trouvées jusqu'à maintenant. Ainsi, à chaque lecture d'une nouvelle lettre `c` de ω :

- on sait, parmi les lettres qui precèdent immédiatement `c`, combien coïncident exactement avec le début du motif
- si `c` est égale à la lettre suivante du motif, alors on saura à la prochaine lecture qu'une lettre de plus a été trouvée
- sinon, la situation est un peu plus compliquée car il faut trouver parmi les lettres qui précèdent `c` combien correspondent à un préfixe du motif en incluant `c`.

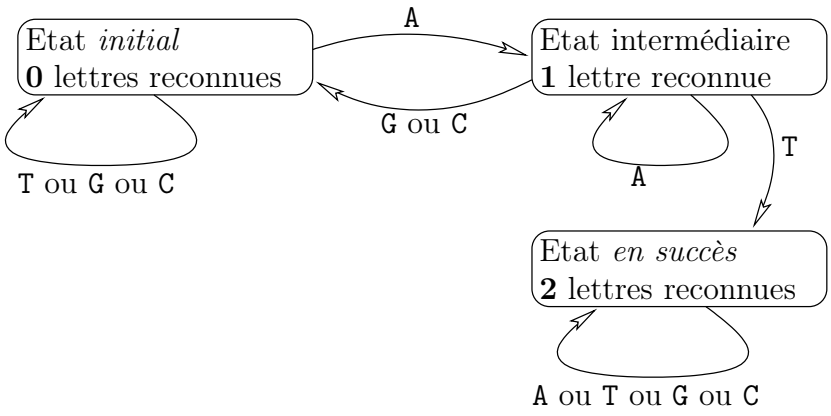
Dans cette dernière étape, si le motif ne contient pas de répétition, alors on retombe souvent sur le préfixe vide. C'est le cas de l'exemple simple du motif `[A;T]`.

- Pour chercher le motif `AT` dans un génome ω, on débute en mémorisant qu'avant de lire la première lettre de ω, aucune lettre du motif `AT` n'a encore été reconnue (bien sûr). On est donc dans un « état initial ».
- Partant de cet *état initial*, si on lit `A` alors on a potentiellement avancé dans la reconnaissance de `AT` : on passe dans un *état* où l'on sait avoir reconnu 1 lettre, c'est-à-dire plus généralement un préfixe de longueur 1 du motif. La prochaine lettre attendue sera donc `T`.
- Si par contre, partant toujours de cet état initial, on ne lit pas `A`, alors on reste dans un état où aucune lettre du motif n'a encore été reconnue. On retourne donc dans l'état initial.
- Partant de l'état où l'on a mémorisé avoir reconnu 1 lettre, si on lit `T` alors on a reconnu 2 lettres du motif et l'on passe dans un *état* où l'on sait avoir reconnu les 2 premières lettres. Comme le motif est de longueur 2, cet état est *en succès* : tout le motif a été reconnu.
- Si par contre, partant de l'état où l'on a mémorisé avoir reconnu 1 lettre, on ne lit pas `T` mais `G` ou `C`, alors tout est à refaire dans la reconnaissance du motif. On retombe donc dans l'état initial, celui où 0 caractères ont été reconnus.
- Enfin si, partant de l'état où l'on a mémorisé avoir reconnu 1 lettre, on lit `A` alors on n'a pas reconnu le motif mais ce `A` peut constituer le début du motif `AT`, on reste donc dans l'état où l'on a reconnu 1 lettre.

Cette façon de voir permet clairement de ne lire qu'une seule fois chaque lettre de ω. En revanche on « paye » cette efficacité par la nécessité de mémoriser combien de lettre du début du motif on a rencontré jusqu'à maintenant. Cette gestion des divers *états* définit *un automate*.

3 Les automates simples

Un automate est défini par un ensemble d'états et des passages d'un état à un autre en fonction des lettres lues. Par exemple, l'automate décrit précédemment pour le motif `AT` est dessiné comme suit :



Par convention, on a estimé que l'on reste dans l'état en succès sitôt qu'on l'a atteint. Si l'on souhaitait continuer le parcours du génome pour trouver *toutes* les occurrences du motif, alors on pourrait passer de l'état en succès à l'état initial pour `T`, `G` ou `C` et de l'état en succès à l'état intermédiaire pour `A`.

Une possibilité pour programmer un automate est de définir la fonction `etatSuivant` qui prend en entrée l'état `e` dans lequel on se trouve et la prochaine lettre lue `c`, et qui retourne en sortie l'état obtenu. Par exemple pour le motif `AT`, cela donne :

```
let etatSuivantAT (e,c) = match (e,c) with
    (0,A) -> 1
  | (0,_) -> 0
  | (1,T) -> 2
  | (1,A) -> 1
  | (1,_) -> 0
  | (2,_) -> 2 ;;
```

Cette fonction définit l'automate pour le motif `AT` de menière tout à fait équivalente au dessin précédent. Elle possède l'inconvénient d'être dédiée au motif `AT` ; on verra en TD comment faire une approche plus générale.

4 Utiliser un automate

Il faut donner en arguments le génome ω mais aussi l'état initial. En effet, s'il est évident que l'état initial sera **0** au début, les appels récursifs au fur et à mesure de l'avancement dans le parcours de ω feront évoluer cet état. L'état en succès pourrait également être donné en argument mais il ne change pas ici : c'est la longueur du motif c'est-à-dire **2**.

```
let rec parcoursAT (e,omega) = match omega with
    [] -> e
  | x::oo -> parcoursAT ((etatSuivantAT(e,x)),oo) ;;
```

On constate que sitôt que le motif est rencontré, la fonction `etatSuivantAT` fournira toujours l'état en succès **2**. On peut donc éviter de continuer le parcours de ω dans ce cas :

```
let rec parcoursAT (e,omega) = if e = 2 then 2
  else match omega with
      [] -> e
    | x::oo -> parcoursAT ((etatSuivantAT(e,x)),oo) ;;
```

Dès lors, la programmation de la recherche de motif devient simple :

```
let motifAT omega = parcoursAT(0,omega) ;;
```

5 Construire un automate

On remplit tout d'abord un tableau dont les colonnes contiennent les états possibles (donc en toute généralité les états de **0** à la longueur du motif), et les lignes contiennent les lettres qui peuvent être lues (donc ici 4 lignes). Dans les cases, on place l'état résultant de la lecture, partant de l'état donné par la colonne.

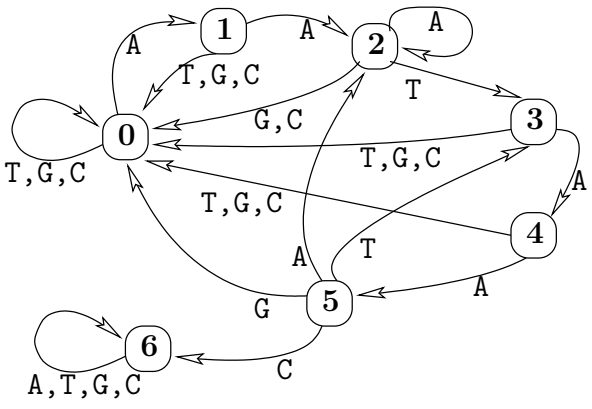
Prenons l'exemple du motif `AATAAC` qui contient une répétition. Considérons par exemple l'état **2** (troisième colonne) : on a reconnu 2 lettres donc le début de motif `AA`

- on attend un `T`, donc si on lit `T` alors on a lu `AAT` et l'on passe dans l'état **3**
- si par contre on lit un `G` ou un `C` alors on a lu `AAG` ou `AAC`, qui ne correspond à rien pour le motif `AATAAC` et l'on retombe dans l'état initial `0`
- enfin si on lit un `A` alors on a lu `AAA` et les 2 derniers `A` peuvent coïncider avec les 2 premiers `A` du motif `AATAAC` ; par conséquent on rejoint l'état **2**.

On fait pareil pour toutes les colonnes :

	0	**1**	**2**	**3**	**4**	**5**	**6**
A	1	2	2	4	5	2	6
T	0	0	3	0	0	3	6
G	0	0	0	0	0	0	6
C	0	0	0	0	0	6	6

La matrice obtenue permet de tracer le graphe de l'automate sans difficulté :



Cours No. 9

1 Recherche de motif exact en général

Ce qui est limitatif dans l'exemple précédent, c'est la fonction `etatSuivantAT` qui dépend du motif `AT`. Pour éviter cela, on va représenter un automate non plus directement sous forme de fonction, mais sous forme de la liste des triplets `(e1,c,e2)` où `e1` est l'état de départ, `c` la lettre lue et `e2` l'état résultant. Par exemple pour le motif `CGTCG` on obtient :

	0	**1**	**2**	**3**	**4**	**5**
A	0	0	0	0	0	0
T	0	0	3	0	0	0
G	0	2	0	0	5	0
C	1	1	1	4	1	1

L'état en succès est traité comme l'état **0**, ce qui permet d'enchaîner les recherches du motif. Ceci donne la liste
`[(0,A,0); (0,T,0); (0,G,0); (0,C,1); (1,A,0); (1,T,0); (1,G,2); (1,C,1); (2,A,0); (2,T,3); (2,G,0); (2,C,1); (3,A,0); (3,T,0); (3,G,0); (3,C,4); (4,A,0); (4,T,0); (4,G,5); (4,C,1); (5,A,0); (5,T,0); (5,G,0); (5,C,1)]`.

On verra en TD comment construire la liste définissant un automate automatiquement à partir d'un motif.

```
# let rec mapadd1 l = match l with
          []  -> []
        | n::r -> (n+1)::(mapadd1 r) ;;
mapadd1 : int list -> int list = <fun>

# let rec cible (tauliste,e,x) = match tauliste with
    [] -> failwith "etat/lettre introuvable!"
  | (e1,c,e2)::r -> if (e1=e) && (c=x)
                    then e2
                    else cible(r,e,x) ;;
cible : ('a * 'b * 'c) list * 'a * 'b -> 'c = <fun>

# let rec listMotif (tauliste,init,succes,omega) =
match omega with
    [] -> []
  | x::oo -> let suiv = cible(tauliste,init,x) in
             if suiv=succes
             then 1::(mapadd1 (listMotif (tauliste,suiv,succes,oo)))
             else mapadd1 (listMotif (tauliste,suiv,succes,oo)) ;;
listMotif : ('a * 'b * 'a) list * 'a * 'a * 'b list -> int list = <fun>

# let tousMotifs (tau,omega) =
  listMotif ((construit tau),0,(list_length tau),omega) ;;
```

On retrouve le typage (plus général que notre cas particulier des nucléotides avec états entiers) et on développe l'exemple d'application de `listMotif` pour le motif `CGTCG` avec la séquence `TGCTCGATCGTCG` en se référant au graphe de l'automate et montrant le parallèle avec les sources de la fonction.

Cours No. 10

1 Alignement de séquences, motifs approchés

Il s'agit ici de déterminer si deux séquences « se ressemblent »...

Cela suppose de définir une notion de distance (≈dissimilarité) entre 2 séquences. Une distance en mathématique est une fonction d qui vérifie, pour toutes séquences ω, ω_i :

- $d(\omega_1, \omega_2) \geq 0$
- $d(\omega_1, \omega_2) = d(\omega_2, \omega_1)$
- $d(\omega, \omega) = 0$
- $d(\omega_1, \omega_2) + d(\omega_2, \omega_3) \geq d(\omega_1, \omega_3)$

Exemple : la distance préfixe,
$d(\omega_1, \omega_2) = |\omega_1| + |\omega_2| - 2 \times |\texttt{prefixmax}(\omega_1, \omega_2)|$.

Exemple : Distance de Hamming = nbre de positions où les séquences diffèrent.

```
let rec hamming (o1,o2) = match (o1,o2) with
    ([],[]) -> 0
  | (_,[]) | ([],_) -> failwith "meme longueur SVP"
  | (x::r1,y::r2) -> if x=y then hamming (r1,r2)
                     else 1 + hamming(r1,r2) ;;

let rec prefixhamming (tau,omega) = match (tau,omega) with
    ([],_) -> 0
  | (_,[]) -> list_length tau
  | (x::r1,y::r2) -> if x=y then hamming (r1,r2)
                     else 1 + hamming(r1,r2) ;;

let rec aligne (seuil,tau,omega) =
match omega with
    [] -> false
  | _::oo -> if prefixhamming (tau,omega) <= seuil
             then true
             else aligne(seuil,tau,oo) ;;

let rec alignements (seuil,tau,omega) =
match omega with
    [] -> []
  | _::oo -> if prefixhamming (tau,omega) <= seuil
              then 1::(mapadd1 (alignements (seuil,tau,oo)))
              else mapadd1 (alignements (seuil,tau,oo)) ;;
```

Autre possibilité :

```
let rec prefixHamming (tau,omega,seuil) =
    if seuil < 0 then false
    else match (tau,omega) with
        ([],_) -> true
      | (_,[]) -> (list_length tau) <= seuil
      | (x::r1,y::r2) -> if x=y
                         then prefixHamming(r1,r2,seuil)
                         else prefixHamming(r1,r2,seuil-1) ;;

let rec alignements (seuil,tau,omega) =
match omega with
    [] -> []
  | _::oo -> if prefixHamming (tau,omega,seuil)
              then 1::(mapadd1 (alignements (seuil,tau,oo)))
              else mapadd1 (alignements (seuil,tau,oo)) ;;
```

Clairement pas assez souple pour un bon alignement. Il vaut mieux une distance en termes d'évolution, donc de mutations élémentaires successives. C'est relativement sophistiqué ; l'idée de base est que les opérations élémentaires sont les suivantes :

- insérer un nucléotide
- supprimer un nucléotide
- remplacer un nucléotide par un autre

et l'on compte 1 de plus dans la distance à chaque fois que l'on doit appliquer une opération élémentaire pour passer de ω_1 à ω_2 (noter que c'est bien une distance en prenant le coût minimum).

Cours No. 11

1 Les fonctions d'ordre supérieur

Le langage ML est fondé sur un principe unificateur qui n'a pas encore été développé dans ce cours : les fonctions sont en fait des données comme les autres. Cela a pour conséquence qu'une fonction peut prendre en argument une autre fonction si on le souhaite. Cela a aussi pour conséquence qu'une fonction peut fournir pour résultat une autre fonction...

En particulier on avait vu deux façons de coder un automate pour la reconnaissance de motif : la première sous la forme d'une fonction et la seconde sous la forme d'une liste de triplets. Ce qui a motivé a l'époque de passer à cette représentation avec des triplet, bien que ce soit plus lourd à gérer, était l'écriture d'une fonction qui prend un motif en entrée et fournit l'automate en sortie (cf. TD numéro 4). Sachant que l'on peut écrire des fonctions « d'ordre supérieur » fournissant une fonction comme résultat, il devient possible de programmer une fonction qui prend un motif en entrée et fournit une fonction représentant l'automate en sortie.

En terme de types de données, c'est en fait très simple :
Si α et β sont des types de données alors ($\alpha \to \beta$) est également un type de données : celui des fonctions ayant α comme ensemble de départ et β comme ensemble d'arrivée.

Plus précisément, rappelons qu'un type de données est défini par le nom du type (ici $\alpha \to \beta$), l'ensemble de ses valeurs, et les fonctions qui travaillent sur ces valeurs. L'ensemble des valeurs de type $\alpha \to \beta$ est l'ensemble des fonctions de la forme

$$\texttt{function}\ x \to \mathit{expr}$$

où x est une variable quelconque et *expr* est une expression de type β du langage ML, dans laquelle x apparaît de type α.
Cela se lit « La fonction qui à x associe *expr.* » et la signification en est claire.

En réalité, à chaque fois que l'on écrit :

```
let f (x1,...,xn) = expression ;;
```

c'est une façon courte d'écrire :

```
let f = function (x1,...,xn) -> expression ;;
```

En réalité, l'ordinateur commence par transformer la première forme en la seconde (qui est la « vraie ») avant de compiler. Ces deux façons d'écrire sont équivalentes ; on peut donc « faire passer » les arguments d'une fonction à droite du « = » avec `function`, ou au contraire supprimer `function` en faisant passer à gauche du « = » les arguments de la fonction.

Enfin les fonctions qui travaillent sur les données de type fonctions (troisième partie indispensable pour définir le type $\alpha \to \beta$) sont en réalité déjà connues de nous : c'est simplement le fait de donner « à manger » des arguments de type α à une fonction pour obtenir une valeur de type β. On sépare simplement la fonction `f` de son argument `x` avec un espace : `(f x)`.

2 Exemples de fonctions d'ordre supérieur

Si l'on écrit par exemple :

```
let double f = function x -> 2 * (f x) ;;
```

alors il s'agit d'une fonction (`double`) qui prend en entrée la variable `f`. Lorsque l'on développe l'arbre de l'expression, on constate que `f` n'est pas une feuille de l'arbre. C'est donc une fonction elle même.
On fait le typage. [`(`α $\to$ `int) ` $\to$ `(`α $\to$ `int)`].

On développe des exemples pour bien comprendre ce que fait cette fonction...

3 Fonctions d'ordre supérieur sur les listes

On se remémore la fonction `mapadd1` et ce qu'elle faisait (ajouter 1 à tous les éléments d'une liste d'entiers). On explique qu'elle applique à chaque élément de la liste une même fonction « `add1` » et qu'il est naturel de vouloir pouvoir le faire avec n'importe quelle fonction.

Explication de ce que fait `map` : $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`, avec exemples...

`map f [a1 ; a2 ;... ; an] = [(f a1) ; (f a2) ;... ; (f an)]`

Ainsi :

```
let mapadd1 l = map (function n -> n+1) l ;;
```

ce qui peut se simplifier en

```
let mapadd1 = map (function n -> n+1) ;;
```

Cette fonction `map` de « peinture par une fonction » d'une liste est fournie par le langage mais on pourrait la programmer soi-même sans difficulté :

```
let rec map f l = match l with [] -> []
  | a::r -> (f a)::(map f r) ;;
```

On remarque que ceci est donc équivalent à :

```
let rec map f = function l -> match l with [] -> []
  | a::r -> (f a)::(map f r) ;;
```

Or, pour simplifier de tels cas, `function` possède un match implicitement disponible. On peut donc écrire de manière plus courte :

```
let rec map f = function [] -> []
  | a::r -> (f a)::(map f r) ;;
```

On remarque que le nom de la liste (`l`) n'apparaît plus.

Remarque appuyée par des exemples sur le fait que dans une expression ML place par défaut les parenthèses à gauche, mais que c'est le contraire (à droite) dans les types. On explique pourquoi tout cela est logique.

À partir des exemples de calcul de la somme des éléments d'une liste, du produit, de la mise à plat d'une liste de listes, etc, on remarque qu'il y a un itérateur qui généralise tout cela.
`list_it` : $(\alpha \to \beta \to \beta) \to \alpha$ `list` $\to \beta \to \beta$
Si l'on fait l'abus de supposer que l'opération binaire du premier argument (type $(\alpha \to \beta \to \beta)$) est infixe, cela revient à remplacer les « ; » par cette opération et à compléter à droite par l'élément `e0` (type β). On obtient ainsi une valeur de type β.
Si `o` est le nom de l'opération binaire, cela donne :
`list_it o [a1 ;a2 ;... ;an] = a1 o (a2 o (... o (an o e0)..))`

Ainsi :

```
let somme = list_it + l 0 ;;
let produit = list_it *. l 1. ;;
let applatir = list_it @ l [] ;;
```

Cet itérateur `list_it` est fourni par le langage mais on pourrait le programmer soi-même sans difficulté :

```
let rec list_it o l e = match l with [] -> e
  | a::r -> o a (list_it o r e) ;;
```

Cours No. 12

1 Passer d'un n-uplet en argument à plusieurs arguments

La curryfication et compagnie.